MXUG #5

An introduction to Parfait

Paul Cowan
Aconex
paul@custardsource.com

July 2009

Laser-Guided Performance Management

Our Problem

(and maybe yours too?)

- Lots of users = *lots* of potential problems
- Big database = *big* problems
- Legacy codebase – not written with performance in mind
- Wildly variable usage patterns
  - Several orders of magnitude difference in amount of data
  - Unpredictable hot spots

Finding the Problems

(Not as easy as you'd think)

- Approach #1: profile hotspots
  - Not always easy (access to data, ability to reproduce, Heisenberg)
- Approach #2: use a simple timing framework
  - Time each request
  - Look for patterns
  - Not always accurate
    - See victims, not causes
    - Not always about wall time!

The Goal

(And the ace up our sleeve)

- Want to get really deep performance metrics
- Export into Performance Co-Pilot
  - OSS framework built @ SGI
  - Collects *lots* of data with low overhead
  - Archive, search, compare patterns, fire alerts...
- Want to easily instrument 3rd-party code

The Brainwave

- Java Webapps have a feature which opens up a world of data
- One request is pinned to one thread for the duration
- And the thread likewise doesn't serve multiple requests
- Whatever we can measure on the thread, we can extrapolate out to the action

How we measure

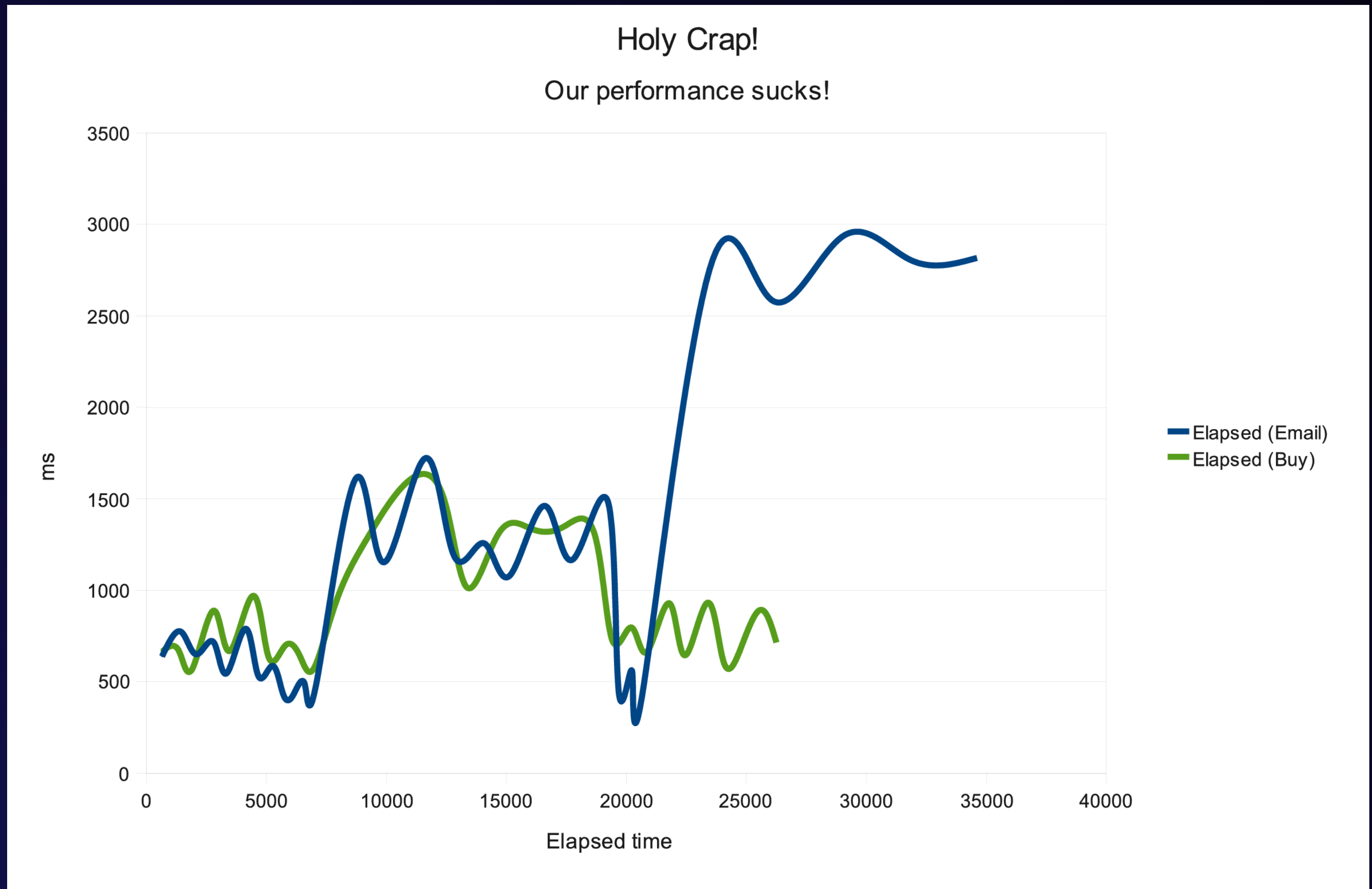
- Have a bunch of per-thread counters
  - Not aware of actions, don't care
- Snapshot values at request start
- Snapshot again at request end
- Delta is that action's "cost"
- Find expensive actions, ~~kill~~ fix them

Built-in sources

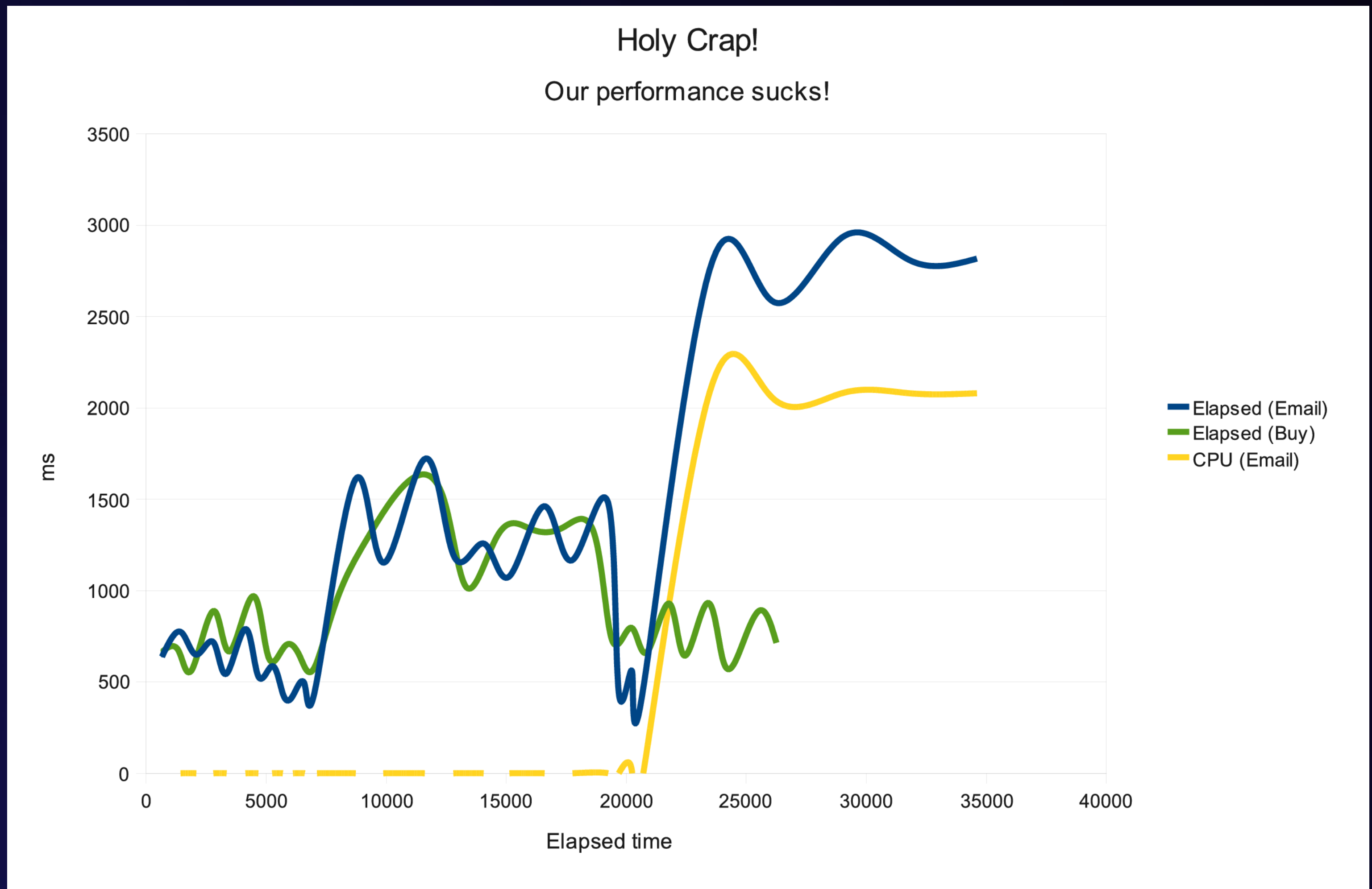
(Here's one they prepared earlier...)

- JVM gives us a bunch of data sources
- `ManagementFactory.getThreadMXBean()`
```
    .getThreadCPUTime(...) /
    .getThreadUserTime(...) /
    .getThreadInfo(...)
      .getWaitedCount() /
      .getWaitedTime()/
      .getBlockedCount() /
      .getBlockedTime()
```
- Suddenly, we can see which user actions are causing contention
- Stats in aggregate, logs for detail
  - Which *user* is eating our CPU?

We've all been here...

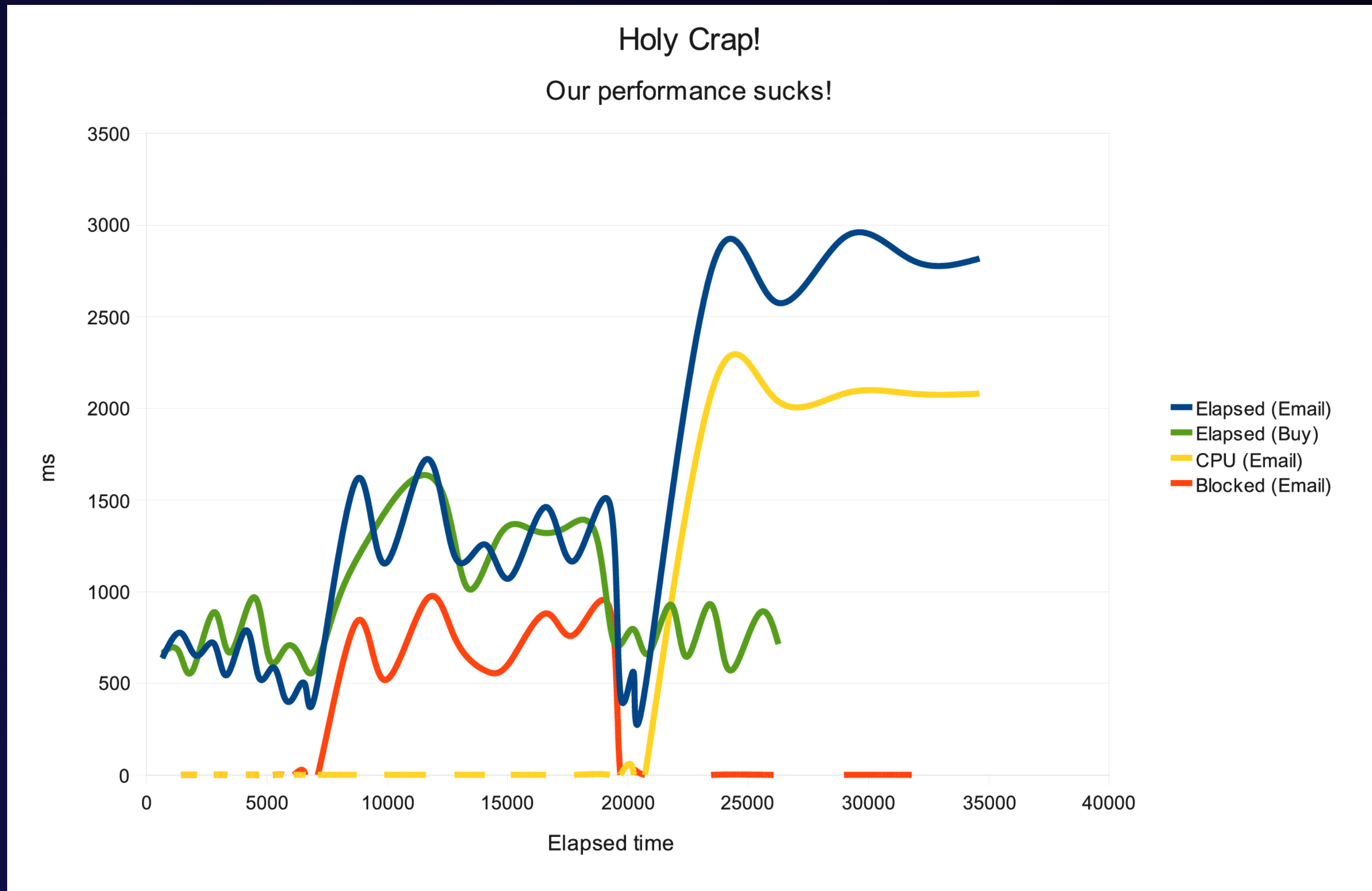


There's more to it...



Mystery not solved

(But at least we know where to look)



EmailSender:sendMail Elapsed time: own 1078ms, total 1078ms Blocked time: own 623ms, total 623ms Wait time: own 455ms, total 455ms User CPU: own 0ms, total 0ms

Adding your own

```
public class StatAppender implements Appender {
 public ThreadLocal<Long> LOG_COUNT = …
 public void doAppend(LoggingEvent e) {
   LOG_COUNT.put(LOG_COUNT.get() + 1);
 }
}
```

Adding your own

```
public class StatAppender implements Appender {
 public ThreadLocal<Long> LOG_COUNT = …
 public void doAppend(LoggingEvent e) {
   LOG_COUNT.put(LOG_COUNT.get() + 1);
 }
}

   … add to log4j.xml, then …

metricSuite.addMetric(
 new AbstractThreadMetric(
 "Log message count", "", "logcount", "…")
 {
   public long getCurrentValue() {
    StatAppender s = (StatAppender)
      Logger.getLogger(…).getAppender("blah");
    return s.LOG_COUNT.get();
   }
 });
```

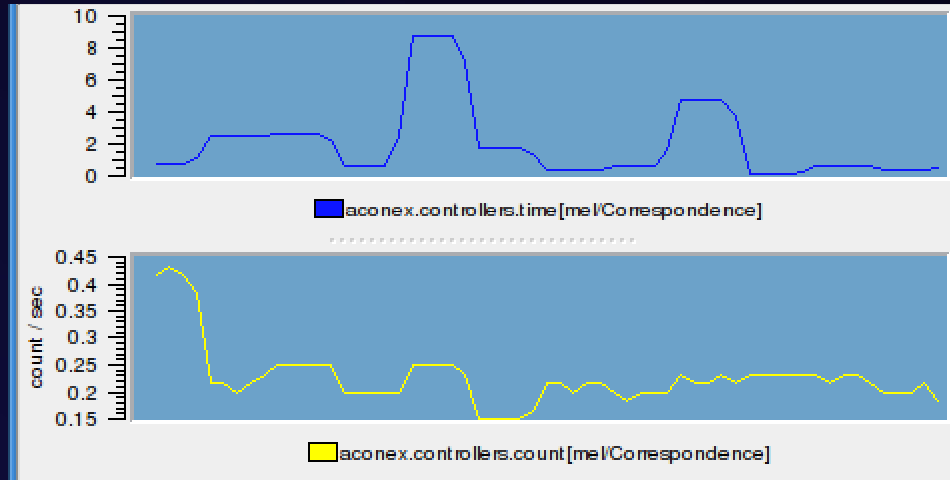
More stuff to measure

(When all you have is a hammer...)

- Custom JDBC driver gives us
  - DB execution counts + times
  - DB physical/logical I/Os
  - DB CPU time (!)
- Error counts through custom error handling mechanism

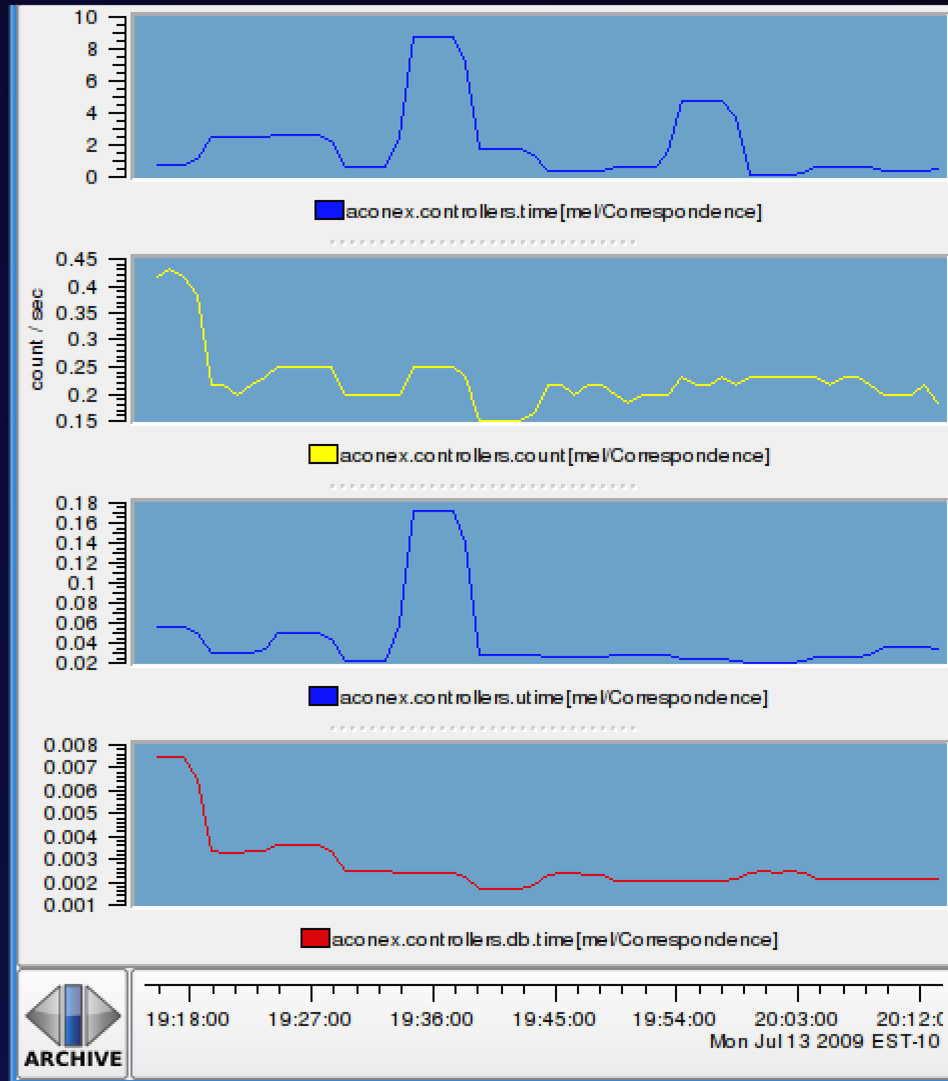
So what do we get?

(Pretty graph time!)



So what do we get?

(Pretty graph time!)



- More metrics:
  - Bytes read/written to client
  - Native library might expose some good OS metrics
- If you can read the current thread, you can read other threads
  - Write our own ‘top’
  - Which threads are hogging resources – and doing what?
- AOP advice – even more transparent
- Deal with background worker threads

Where's it going?
(And can I get on board, man?)

- Now open-sourced as ‘Parfait’
- Includes general PCP library (dxm)
- Modular (not tied to any input/output mechanism)
- Lives on Google Code
- Hungry for users and contributors!